You CUDA Had It All: Object Oriented Fortran and Porting to GPUs

Jenniffer Estrada[§*] *Mentors: Joseph Schoonover*, Bob Robey**

[§] *Youngstown State University* [*] *Los Alamos National Laboratory*

LA-UR-16-25634

Abstract

We demonstrate performance metrics of an Object Oriented Fortran code with GPU acceleration enabled through NVIDIA/PGI's CUDA on different GPU architectures. PGI's support of Obejct Oriented Fortran could enable a wider range of scientific problems to be accessible on fewer resources by leveraging the power of a graphics card. As an example, spectral element methods have been shown to be arithmetically intensive, suggesting significant speedup can result from parallelization on GPUs. An open source Object Oriented Fortran spectral element library, SELF, is parallelized using CUDA and OpenMP. Code performance is examined using a Tesla M2090 (Fermi), GeForce GTX Titan X (Maxwell), Tesla K20, and Tesla K40 (Kepler) with the cuBLAS library and different code restructuring to play to the strengths of GPUs.

What is SELF?

– Spectral Element Libraries in Fortran (SELF) is a collection of tools for implementing continuous and discontinuous Galerkin spectral element methods in 2-D and 3-D.

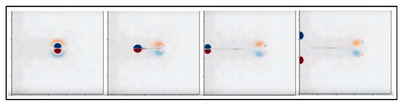


Figure 1: SELF-Shallow water solver demonstrating dipole propagation

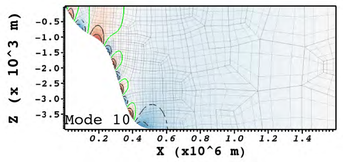


Figure 2: The pressure field associated with a topographic wave mode that is extracted using the SELF-CGSEM libraries. Topographic waves help explain ocean dynamics on continental margins and may help explain processes like the Gulf Stream separation *(Stern, 1998)*

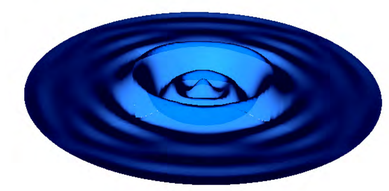


Figure 3: Shallow Water Visualization

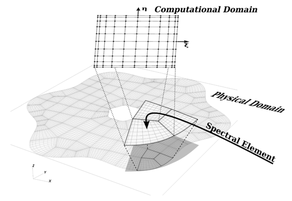


Figure 4: Illustration that the spectral element method can result in many dense matrix-matrix operations.

Spectral Elements

– The global physical domain is decomposed into "spectral elements". Within each spectral element, functions are approximated by polynomial interpolants of arbitrary order.

– By mapping each element from physical space to a reference computational space, the algorithm can be written to exploit SIMD parallelism.

–Through interpolation formulas, derivative operations (divergence, gradient, curl) are expressed within each element as matrix-matrix products.

–The first matrix is a "derivative-matrix" and is the same for each element. The second matrix typically involves combinations of the PDE solution and metric terms; hence, the data for the second matrix in the product varies with each element.

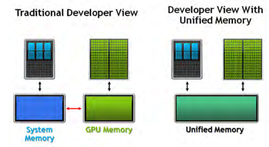
– 2-D problems typically require between 50MB and 100MB during a simulation for total memory usage, and 3-D problems are expected to tackle systems with $O(10^6 - 10^8)$ degrees of freedom, giving memory requirements between 500 MB and 1GB.

– *The small memory footprints and high algorithmic intensity make SELF an ideal candidate for porting to GPUs.*

OpenMP, OpenACC and CUDA

– OpenACC has very similar constructs to OpenMP and can be added to codes that already have OpenMP threads.

– CUDA and OpenACC can take advantage of Unified Memory Management, which has the CPU and GPU seeing the same memory with the use of pointers.



Limitations

– Data movement is costly (start-up cost and cost per byte), so the aim is to move the data infrequently. PCI bus is great as an I/O bus, but terrible as a memory bus. Ideally, minimize frequency and volume, and increase the regularity of the data (contiguous data).

– Small cache, but latency insensitive.

– Complex data structures are not easily ported and have limited support with CUDA or OpenACC, but Unified Memory Management makes this easier to handle.

Methodology

– Spectral element methods have been shown to be arithmetically intensive, and can greatly benefit from parallelization on a GPU (*Abdi, 2016 (NUMA GPU Implementation)*).

– Running the code under Allinea-MAP profiler and generating a callgraph with Valgrind's Cachegrind, it was shown that the most time consumed was during several matrix matrix multiplication operations, with MappedTimeDerivative subroutine taking the longest with respect to the total execution time.

– The scaling of the matrix-matrix multiplication on the CPU sees a linear increase in wall time, compared to the GPU wall time logarithmically increasing with dimensionality.

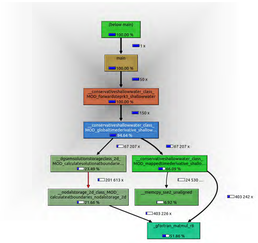


Figure 5: Call Graph of SELF

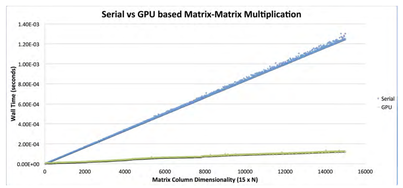


Figure 6: Serial *(Fortran90 compiled with PGI on Eight-Core Intel Xeon model E5-2670 @ 2.6 GHz)* vs GPU *(CUDA Fortran compiled with PGI on Tesla M2090)* parallelized matrix multiplication

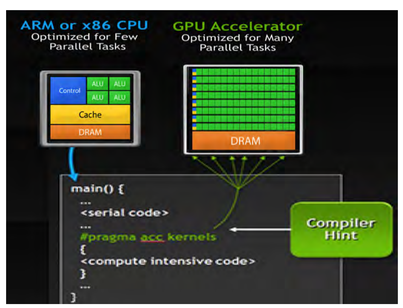
Algorithm Optimizations

– The original code calculated several matrix multiplications using small square matrices

– Due to the high cost of moving data to and from the GPU, the small matrices are concatenated together to be sent to the GPU and do the multiplication, and then separated out after being returned.

– CUDA C and the double precision matrix-matrix multiplication routine from the CuBLAS library is used.

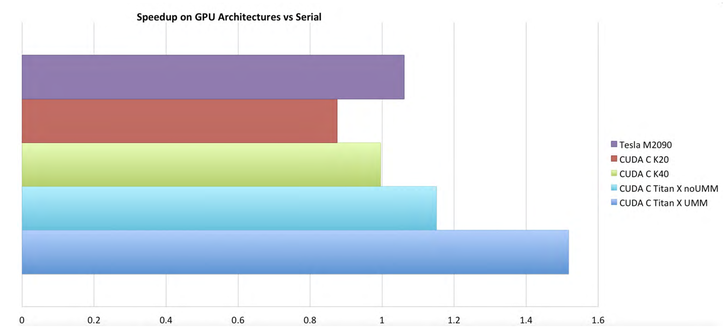
– To reduce the data transfers, memory is allocated once on the GPU, and all static variables are passed once. The flux divergence matrix is the only data changing and being copied with every time step.



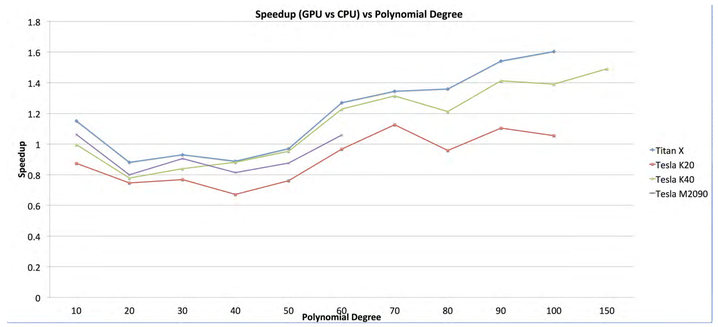
– Porting greater portions of SELF is necessary for significant performance improvements, which is feasible with OpenACC and CUDA.

Results

CUDA



– There is a slight speedup in performance with the Tesla M2090, which might be due to the influence of the underlying CPU and bus, compared to the Darwin nodes. On the Kepler architecture there is no difference on the K40 and slower performance on the K20. There is a 1.5x speedup with the Titan X using Unified Memory management with CUDA C and the cuBLAS library (NVCC compiler) cross-compiled with Fortran (PGI compiler) compared to the serial Fortran version. There is a noticeable positive effect with using Unified Memory Management with CUDA.



– Varying the polynomial degree of of the shallow water equations, causing greater matrix sizes demonstrates that there is a theoretical performance improvement with larger problems on the GPU.

Conclusion

– The advantage of using GPUs for computational problems that lend themselves to high arithmetic intensity and parallelization is evident, but porting FORTRAN codes using newer standards using Nvidia's CUDA programming language is not currently feasible without massive re-writes and a need for writing specific kernels to accomplish handling the derived types on the GPU more efficiently, whereas OpenACC requires less of an effort to port existing FORTRAN codes to GPUs using compiler directives, but memory management might not be optimal due to being left completely up to the compiler.

– Utilizing and optimizing GPU parallelization and heterogeneous platforms in physics applications is numerically feasible on unstructured meshes, and offers new avenues for the future of computational sciences as we approach the age of exa-scale high performance computing.

– Application submitted to ORNLHack, a 5 day GPU Hackathon at Oak Ridge in October, to finish implementation of GPU acceleration.

References and Acknowledgements

M.E. Stern, 1998, *Separation of a density current from the bottom of a continental shelf*, J. Phys. Oceanogr., 28, 2040–2049

For 3D NUMA GPU implementation, see:

Abdi et al. *GPU Accelerated Spectral Element Methods: 3D Euler equations.* Naval Post Graduate School, 2016.

For more information on SELF and the Spectral Element Methods code, see:

https://github.com/schoonovernumerics/SELF

Special thanks to Jeff Larkin, *NVIDIA*